GRAPH ALGORITHMS

https://doi.org/10.5281/zenodo.6502327

Khushbakov Sherzod
Student of TUIT FSE
Khamraev Mansur
Student of TUIT FSE
Bakhtiyorova Mokhiruy
Student of TUIT FSE,

Abstract: *Graphs are widely-used mathematical structures visualized by two basic components: nodes and edges. Graph algorithms are used to solve the problems of representing graphs as networks like airline flights, how the Internet is connected, or social network connectivity on Facebook. This article will introduce you to the tools you need to master important graph algorithms and the optimal search algorithm in Python.*

In terms of computer science and discrete mathematics, graphs are an abstract way of representing types of relationships, such as roads connecting cities and other kinds of networks. Graphs are made up of edges and vertices. A vertex is a point on a graph, and an edge is something that connects two points on a graph.

The terms of graph theory problems in programming competitions usually talk about things like networks and lattices.

Here is a list of all graph theory terms you need to know:

• A path—a sequence of edges connecting different (non-repeating) vertices;

• Routes—these are the same paths, only they do not require a sequence of different vertices;

• Cycle—a group of vertices connected together in a closed chain. In the figure above, the vertices [1, 2, 4] form a cycle;

• A connected graph—graph in which there is one path between any pair of vertices;

• A tree is a connected graph that does not contain a cycle;

• An undirected graph—a graph in which the edges have no direction. The figure above shows just an undirected graph. In such an undirected graph, one can move along an edge in either of two directions;

• A directed graph—a graph in which the edges have directions and are denoted by arrows. In such a directed graph, you can move along the edge only in the specified direction.

Adjacency matrices

The adjacency matrix is a graph in the form of a two-dimensional matrix with dimensions $V\ x\ V$, where V is the number of graph vertices. Adjacency matrices are best applied when V^2 is approximately equal to E (number of edges), i.e. when the graph is dense. The entry a_ij indicates how many edges connect vertex i and vertex j.

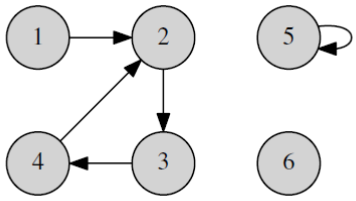


Figure 1. Directed graph.

Directed graph, its adjacency matrix is shown below:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A 6 x 6 matrix is representing the directed graph above.

Choosing the optimal search algorithm in Python

When it comes to learning, we tend to take one of two main approaches: either go broad and try to cover as much of the area as possible, or deep and try to get specific about the topic we are studying. Successful people note that everything we study in life, from algorithms to life skills, to a certain extent implies a combination of these approaches.

Here we will get acquainted with the two main search algorithms, namely depth-first search (DFS) and breadth-first search (BFS), which will form the basis for understanding more complex algorithms.

Content:

1. Tree traversal.
2. Search in depth.
3. Breadth first search.
4. Comparison of the proposed algorithms.

Let's start by traversing the tree.

Tree traversal is commonly known as checking (visiting) or updating each node once without repetition. Since all nodes are connected by edges, we always start from the root node. This means that you cannot arbitrarily refer to any node in the tree.

There are three approaches to bypassing:

- straight;
- symmetrical;
- reverse.

Direct Bypass

In this method, we first read data from the root node, then move to the left subtree, and then to the right. Because of this, the nodes we visit (as well as their data output) follow the same pattern in which we print the root node data first, then its left subtree data, and then the right one.

Algorithm:

Until all nodes have been visited

Step 1 – Visiting the Root Node

Step 2 – Recursive traversal of the left subtree

Step 3 – Recursive traversal of the right subtree

Preorder: root – left – right;

Read the data at root first -> then go left -> then go right;

1 -> root node

2, 3, 4 -> left subtree

5, 6, 7 –> right subtree

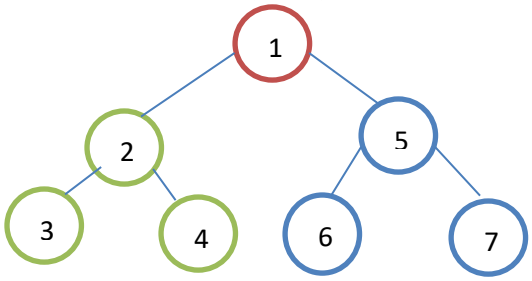


Figure 2. Direct Bypass.

We start at the root node and, following the direct traversal order, first visit the node itself, and then move on to its left subtree, which we traverse in the same way. This continues until all nodes have been visited. As a result, the output order will be: 1,2,3,4,5,6,7.

Symmetric Bypass

With symmetrical traversal, we traverse the path to the leftmost child, then return to the root, visit it, and follow the path to the right child.

Algorithm:

Until all nodes have been visited

Step 1 – Recursive traversal of the left subtree

Step 2 - Visiting the Root Node

Step 3 – Recursive traversal of the right subtree

Symmetric Bypass

Starting at root node 4, we recursively iterate over its left subtree using the same symmetrical order, then visit the root node itself, and then iterate over its right subtree.

Reverse bypass

In the reverse approach, we first visit the left child, then the right one, and after completing the traversal of the subtrees, we read the root.

Algorithm:

Until all nodes have been visited

Step 1 – Recursive traversal of the left subtree

Step 2 – Recursive traversal of the right subtree
Step 3 – Visiting the Root Node

- Postorder – left – right – root;
- Read anything from the left first;
- Then from the right note;
- Then read data at the root;

7 -> root node
1, 2, 3 -> left subtree
4, 5, 6 –> right subtree

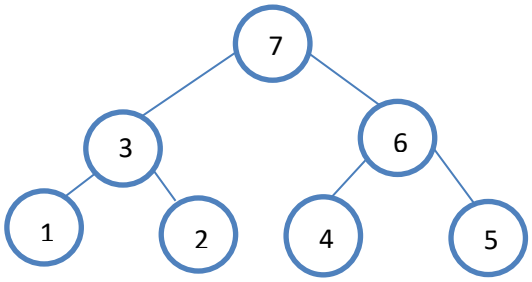


Figure 3. Reverse bypass.

It becomes clear that algorithms are classified based on the sequence in which nodes are visited.

Here I will mention again that there are two main techniques that we can use to traverse and visit each node exactly once: depth-first search or breadth-first search.

Depth First Search (DFS)

In this way, we always visit the deepest node, then go back and follow another path, reaching another end node.

Note that this algorithm uses the stack mechanism to remember the route to and from the end node.

With this approach, we need to traverse the entire branch of the tree and all adjacent nodes. Therefore, keeping track of the current node requires a last-in-first-out approach, which is implemented via a stack. Once the deepest node is reached, all other nodes are popped from the stack. It then traverses adjacent nodes that have not yet been visited.

If a queue were used instead of a stack, representing a first-in-first-out approach, then we would not be able to go deep without removing the current node from the queue.

Stack: last in, first out

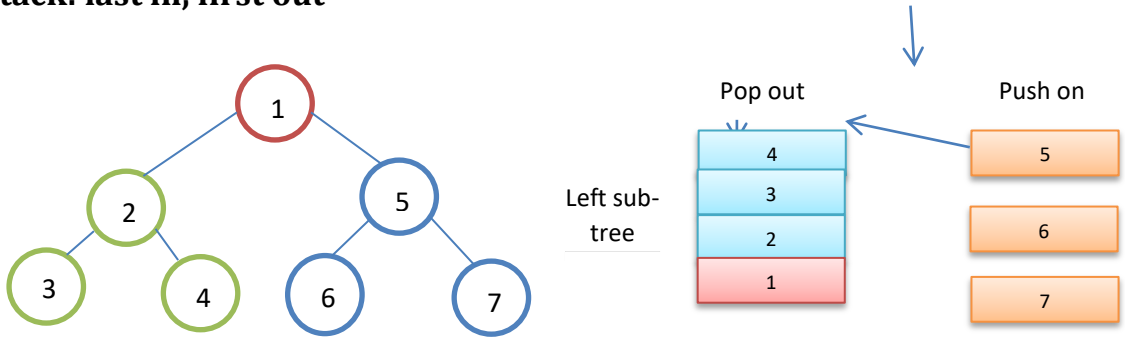


Figure 4. DFS

Hiking DFS is like walking through a maze. You explore the path until you reach its end point, after which you return and follow another.

DFS in Python

We represent the tree in code using an adjacency list through a Python dictionary. For each vertex there is a list of nodes adjacent to it.

```
graph = {
'A' : ['B','C'],
'B' : ['D', 'E'],
'C' : [],
'D' : [],
'E' : []
}
```

Next, we define visited node tracking through the visited = set() statement.

Based on the adjacency list and starting at node A, we can find all the nodes in the tree using the recursive DFS function. dfs function algorithm:

1. Check if the current node has been visited. If yes, then it is added to the appropriate set.

2. The function is called repeatedly for each neighbor of the node.

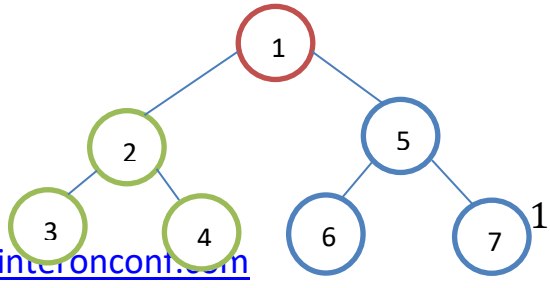
3. The base case is called when all nodes have already been visited, and after that the function returns.

```
def dfs(visited, graph, node):
if node not in visited:
print (node)
visited.add(node)
for neighbor in graph[node]:
dfs(visited, graph, neighbor)
>>>dfs(visited,graph,'A')
```

Breadth First Search (BFS)

In this approach, we search through all nodes of the tree, creating a wide network. This means that we first bypass one level of descendants and only then move on to the next level of their descendants.

Such a search first examines the nearest nodes and then moves further and further away from the starting point. With this in mind, we want to work with a data structure that yields the oldest element when needed; counting them in the order they were added. Here we need to apply the “queue” mechanism.



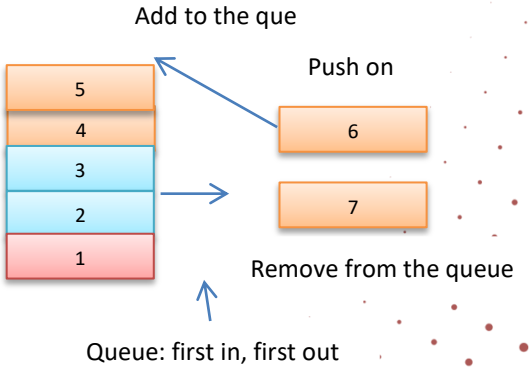


Figure 5. BFS.

BFS in python

We also represent the tree in code using an adjacency list via a Python dictionary. Each vertex stores a list of nodes adjacent to it.

```
graph = {
'A' : ['B','C'],
'B' : ['D', 'E'],
'C' : [],
'D' : [],
'E' : []
}
```

Next, to keep track of visited nodes, we set visited = [].

To keep track of the nodes that are in the queue, we set queue = [].

Given an adjacency list and starting from node A, we can find all nodes in the tree using the BFS recursive function, which:

1. First checks and adds the start node to the visited list and also to the queue.
2. Further, while there are elements in the queue, it continues to exclude nodes, add their unvisited neighbors, and then mark them as visited.
3. Performs these actions until the queue is empty.

```
def bfs(visited, graph, node):
visited.append(node)
queue.append(node)

while queue:
s = queue.pop(0)
print (s, end = " ")
for neighbor in graph[s]:
if neighbor not in visited:
visited.append(neighbor)
queue.append(neighbor)
```

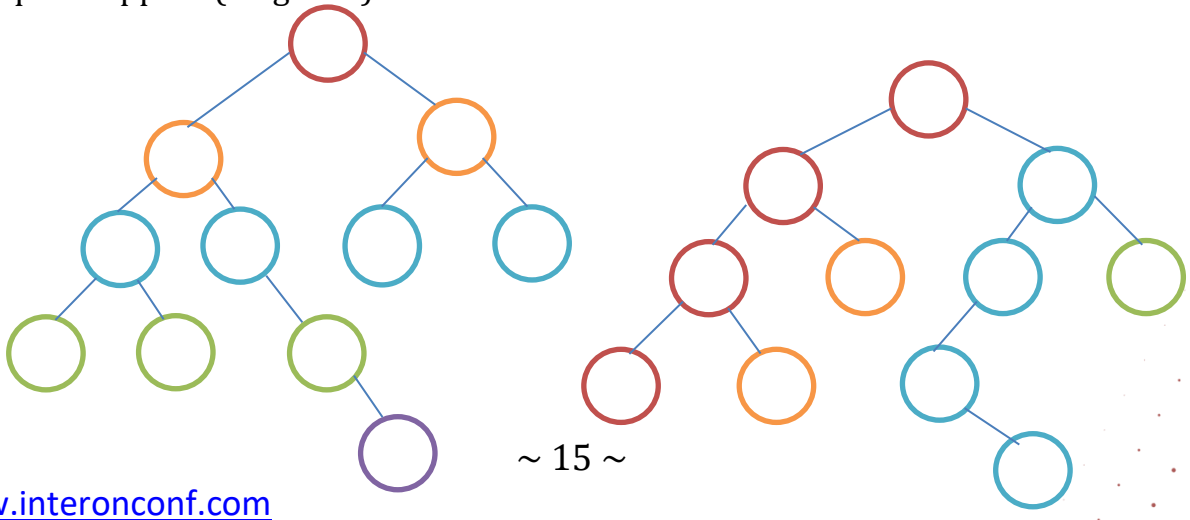


Figure 6. BFS and DFS.

So we looked at the differences between DFS and BFS. You are probably also interested in knowing when and which one is best. At an early stage of studying algorithms, I also asked myself this question. I hope my answer can provide sufficient clarification:

• If we know that the desired point is not far from the root, then it is better to use BFS.

• If the tree has a very deep structure, and the required points in it are rare, then DFS can take a very long time. BFS will do it faster.

• If the tree is very wide, then BFS may require so much memory that it is not practical.

• If the points you are looking for occur frequently but are located deep in the tree, BFS may also be impractical.

Conclusion:

Usually you should use:

• BFS, when you need to find the shortest path from a particular source node to a desired point. In other words, when we are interested in the path with the least number of steps leading from a given initial state to the desired one.

• DFS, when you need to explore all the possibilities and find the best or recalculate the number of possible paths.

• BFS or DFS, when we only need to check the existence of a connection between two nodes of the presented graph, or, in other words, to find out if we can reach one while being in the other.

REFERENCES:

1. https://nuancesprog.ru/p/9269/
2. https://nuancesprog.ru/p/9284
3. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
4. https://web.stanford.edu/class/cs97si/06-basic-graph-algorithms.pdf